Ludovic GUÉGAN
Master LID – Option Image

Mémoire de Master :

Hybridation d'agents pour systèmes embarqués

Encadrement :
Serge Stinckwich (Équipe MAD – Greyc)
Noury Bouraqadi (École des mines des Douai)
Responsable :
Christine Porquet (ENSICAEN)

Résumé

Dans un système embarqué les ressources sont un sujet de contrainte pour le développeur. En effet elles peuvent évoluer au cours du cycle de vie du système embarqué ou différer d'une application à l'autre ce qui exige du développeur à prendre en compte les ressources. Une solution pour palier à cette contrainte est de construire un agent adaptable aux ressources. Alors que ce problème est traditionnellement algorithmique, il est maintenant possible de le traiter du point de vue génie logiciel ; grâce à la programmation par aspect. Ceci est l'objet du stage présenté.

Table des matières

Chapitre 1

Présentation et problématique

1.1 À propos du sujet : Contexte

Cette étude à pour objectif de permettre le contrôle de l'usage des ressources dans les systèmes embarqués. Ainsi nous allons voir en quoi les systèmes multi-agents peuvent permettre de répondre aux problématiques rencontrés par les systèmes embarqués. Mais tout d'abord il nous faut expliciter brièvement ces notions avant de définir la problématique du sujet.

Systèmes embarqués Les systèmes embarqués sont des équipements spécialisés intégrés dans de plus vastes systèmes. On peut citer à titre d'exemple les routeurs de l'Internet, Les téléphones portables et les systèmes de conduite assistés. Tout ces équipements partagent une caractéristique : leurs ressources sont limitées. Ainsi les ressources matérielles disponibles sont une contrainte critique dans le développement d'applications embarquées.

Ressources On appelle ressource toute grandeur consommable par l'activité de l'agent. Par exemple le nombre de cycles du processeur monopolisés pour effectuer une décision, la quantité de mémoire allouée lors d'un calcul ou encore la bande passante consacrée au transfert d'un fichier. La notion de ressource ne sera pas définie de façon formelle. Les ressources considérés dépendent du contexte. Par exemple, une ressource peut être un composant singleton doué d'un état.

Système multi-agents Un système multi-agents (SMA) est une organisation distribuée d'entités autonomes appelées agents. Les caractéristiques d'un SMA reposent principalement sur celles de ces agents. Voici quelques caractéristiques que peuvent posséder les agents :

- Autonomie : Les agents sont capables de réaliser la plupart de leurs tâches sans assistance extérieure.
- Réponses et Réactivité : Les agents sont capables de répondre dans des temps adaptés aux événements extérieurs.
- Comportement actif et délibération : Le comportement de l'agent est guidé par un but, c'est-à-dire que l'agent ne réagit pas seulement à son environnement, mais il est capable de

délibération pour choisir de façon rationnelle ses actions.
- Collaboration : L'agent est capable d'interagir (donc de communiquer) avec d'autres entités pour résoudre un problème.
- Adaptation : L'agent est capable de modifier son comportement en fonction de son environnement et de ses ressources.

En concervant ces propriétés, nous allons étudier comment on peut adapter un agent pour un systèmes embarqués dont les ressources sont limités.

Coût de fonctionnement On définit le coût d'une activité en fonction de l'usage qui est fait des ressources. Nous n'allons pas essayer de définir une mesure d'usage de toutes les ressources disponibles (ex : énergie restante, mémoire disponible, bande passante accessible). Nous supposons simplement que de telles mesures existent[1]. À partir de cette notion de coût il est possible de définir le coût d'exécution d'un agent comme la somme de deux coûts :
- le coût de décision des actions à entreprendre
- le coût d'exécution de ces actions

Par la suite nous ne nous intéresserons qu'au coût de décision. C'est ce coût qui va être étudié dans la présentation des divers modèles d'agents.

1.2 Modèles d'agents

1.2.1 Classification des architectures

Il existe plusieurs critères qui permettent de différencier les modèles d'agents mais nous utilisons uniquement le *processus de décision* de l'agent. Il s'inscrit dans le cycle perception – décision – action de l'agent (voir figure 1.1). Le processus de décision concerne tout ce qui relie les perceptions aux actions ; on parle aussi d'architecture de contrôle. Deux tendances se distinguent : les architectures réactives où la décision se fait sans utiliser de connaissances sur le contexte et sans en garder de trace et les architectures délibératives qui font appels à des connaissances dans leur processus de décision. Enfin, il existe des architectures qui mêlent ces deux caractéristiques : ce sont les architectures hybrides.

1.2.2 Agents réactifs

Les modèles d'agents réactifs possèdent des "comportements" qui vont être exécutés en fonction des situations rencontrées par l'agent. Ces comportements sont généralement enregistrés dans l'agent qui n'a qu'à les restituer au momment adéquat. Les comportements sont associés à des conditions d'exécution. Le choix d'un comportement en cas de conflit est alors une tâche délicate.

[1] Plus de détails sur la mesure de l'utilisation d'une ressource peuvent être trouvés dans (Rus91).

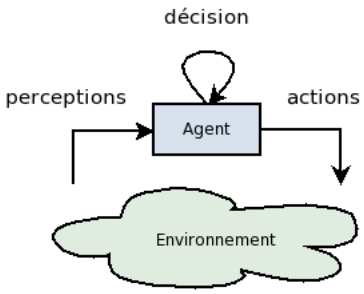

Fig. 1.1 – Représentation symbolique de l'activité d'un agent

Nous allons présenter deux solutions à ce problème : Tout d'abord une architecture "descendante" appelée *subsomption*, puis une évolution de cette architecture.

Architecture de type subsomption Proposée par Brooks en 1986 (Bro86), cette architecture repose sur un ensemble de comportements ordonnés par priorité. À chaque cycle (voir figure 1.1) tout les comportements sont testés en parallèles. Le système sélectionne le comportement de plus haute priorité. Les comportements sont susceptibles de partager des informations et d'inhiber des comportements de priorité inférieure. Ce système à l'avantage d'être simple et très réactif, il permet d'effectuer plusieurs actions simultanément. Cependant l'objectif de la mission n'est pas explicite et lorsqu'il comporte beaucoup de comportements l'exécution du système est difficile à prévoir.

Architecture DAMN Proposé par Rosenblatt en 1995 (Rose95), cette architecture fonctionne sur le même principe qu'une architecture de type subsomption à la différence que les comportements y sont élus. Chaque comportement évalue sa propre pertinence et en informe un ensemble d'arbitres. Ensuite les arbitres sélectionnent les comportements en fonction de leurs propres préoccupations. L'inconvénient de cette approche est qu'il n'y a pas d'interactions entre les comportements : Il n'est pas permit au comportement de partager des calculs ou de s'inter-activer par exemple. De plus les comportements sont activés à la manière de neurones par effet de pallier en fonction de poids. il faut donc ajuster les poids des comportements pour obtenir le résultat recherché. Il est donc difficile de contrôler précisément l'agent lorsque le système s'exécute avec beaucoup de comportements.

1.2.3 Agents délibératifs

Les agents délibératifs possèdent un état, appelé état mental. Cet état mental contient généralement une représentation de l'environnement de l'agent[2]. Cette représentation permet de prévoir

[2]Cette représentation peut être explicite ou implicite. Dans tout les cas l'agent fera preuve de "mémoire" dans sa décision.

ou d'anticiper les états futurs du monde. Ainsi l'agent va pouvoir agir de façon cohérente dans le temps. C'est la principale différence avec les architectures réactives.

Architecture BDI : Belief, Desire, Intention Dans une approche traditionnelle de l'intelligence artificielle, un agent possède un "état mental" qui évolue en fonction d'une représentation symbolique qu'il a de son environnement : L' architecture BDI (Beliefs, Desires, Intentions) décrit un tel modèle. Beaucoup d'architectures reposent sur ce modèle (détaillé dans (Rao95)) qui formalise les structures cognitives de l'agent. Selon ce modèle l'agent possède des désirs (objectifs) qu'il va mettre en œuvre par des intentions (plan d'actions) en fonction de ses croyances (connaissances).

- Une *croyance* décrit une attente sur l'état du monde. L'ensemble des croyances forme la représentation du monde d'un agent.
- Un *désir* décrit une préférence sur l'état futur du monde (fait de croyances). Ces désirs ne sont pas nécessairement réalisables et peuvent s'opposer.
- Les *intentions* représentent les actions qu'un agent souhaite entreprendre. Elles diffèrent des désirs qui ne sont pas toujours compatibles ou réalisables en même temps.

À cause de la généralité de ces définitions on fait appel aux notions plus pragmatiques que sont les situations, les buts et les plans.

- Une *situation* représente la réalisation particulière d'un cas d'exécution connu. Les situations sont utilisées comme des déclencheurs.
- Un *but* représente les désirs réalisables d'un agent. Ils permettront de définir ces intentions.
- Un *plan* représente la mise en œuvre pratique d'une intention et s'exprime dans un langage de programmation.

Nous allons voir par la suite qu'InteRRaP intègre différemment les notions de situation, de but et de plan selon les couches (réactive, délibératives et collaboratives).

planification La planification est une composante importante des architectures délibératives. En effet, l'explosion combinatoire des possibilités limite la planification que les agents sont capables de faire. Il est nécessaire de faire appel à des algorithmes qui permetent de planifier une séquence d'actions dans un temps "raisonnable". La planification a donc un coût important en terme de calcul et de mémoire qu'il faut minimiser.

1.2.4 Agents hybrides

Les agents hybrides essaient de bénéficier des avantages des deux approches sans en garder les inconvénients. Voici un récapitulatif :

- architecture réactive :
  avantage : forte réactivité au contexte, faible coût de décision
  inconvénient : manque de cohérence du comportement global
- architecture délibérative :

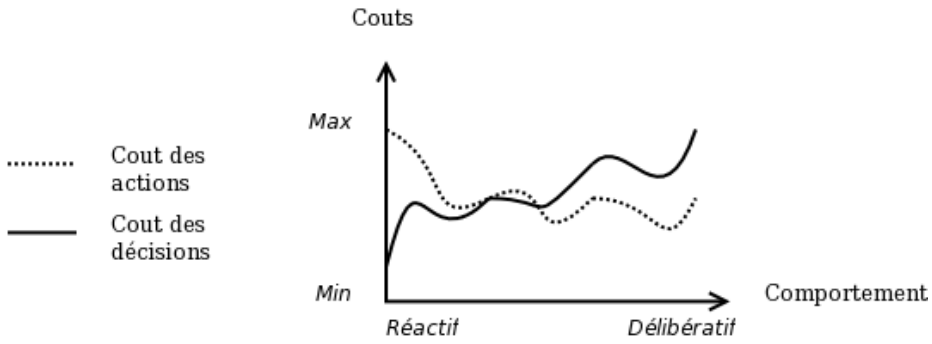

Fig. 1.2 – Les coûts de fonctionnement

avantage : comportement guidé par des intentions
inconvénient : manque de réactivité, coût de décision élevé
Pour y parvenir, les agents hybrides associent les deux approches dans une architecture généralement en "couches". L'agent est alors constitué de plusieurs modules qui ont des rôles spécifiques (comme par exemple la gestion d'un comportement réactif, une capacité inductive d'une base de connaissance, un planificateur de taches). Ces architectures sont difficilement unifiables autour d'un unique principe de fonctionnement. Elles résultent plutôt de l'assemblage de composants.

1.2.5 Bilan des architectures

Les diverses architectures exposées diffèrent principalement par leur processus de décision. Les ressources allouées à cette tâche varies également. Cependant les architectures hybridse proposent une hybridation "câblée" dans le modèle. C'est à dire des modèles plus ou moins réactifs ou délibératifs soit hybrides, mais dont on ne contrôle pas la consommation des ressources par le processus de décision.

1.3 Formulation du problème

Les systèmes embarqués ne disposent que de faibles ressources. D'autre part les modèles d'agents proposés ne permettent pas de choisir le comportement de l'agent en fonction d'une utilisation des ressources. La figure 1.2 illustre la distinction entre le coÃżt du processus de décision et celui des actions effectuées. l'objectif de ce travail est de permettre au concepteur de varier le coÃżt des décisions de l'agent en fonction de ses besoins et des ses ressources. Il s'agit de permettre de rendre l'hybridation paramétrable.

Mise en garde Il est évident que les approches réactives et délibératives ne répondent pas aux mêmes problématiques. Un comportement délibératif n'est pas *dans la lignée* d'un comportement réactif. Ces deux attitudes ce complètent plus qu'elles ne s'opposent. Cependant on constate

qu'une approche "plutôt réactive" convient mieux lorsque les ressources manquent alors qu'au contraire, lorsque l'environnement de destination est "calme" il est avantageux de concevoir un agent qui saura "réfléchir". Nous allons décrire dans la section suivante le modèle d'agent hybride appellé InteRRaP.

1.4 InteRRaP : Un modèle d'agent hybride

1.4.1 Aperçu

InteRRaP est un modèle d'agents hybrides créé par Jörg P.Muller. L'objectif d'InteRRaP (Intégration of Reactivity and Rational Planning) est de concilier les approches réactive et délibérative ainsi qu'une approche coopérative qui met en jeu plusieurs agents dans la réalisation d'une action. Pour ce faire InteRRaP propose la séparation suivante des comportements :

1. Une couche réactive
2. Une couche délibérative
3. Une couche coopérative

Ces trois couches fonctionnent sur le même modèle. Il s'agit d'un cycle d'exécution basé sur le modèle "perception – décision – action" dont voici les principales étapes : mettre à jour la représentation du monde, modifier les motivations et déterminer de nouvelles intentions pour enfin les mettre en œuvre. La figure **??** illustre les interactions entre les couches.

InteRRaP distingue trois activités qui correspondent aux trois couches : les réflexes, les reflexions et les collaborations. D'autre part InteRRaP est une architecture BdI. L'activité interne d'un agent peut se résumer par un *cycle de contrôle* représenté figure 1.4. Ainsi chaque couche fonctionne de faÃğon globalement identique.

Propriétés générales Les comportements sont divisés selon leur nature réactive, délibérative ou collaborative. Cette séparation structurelle simplifie la conception de l'agent. Cette décomposition en couches et surtout les interactions prévues entre ces couches permettent de répondre aux problématiques propres à la conception de systèmes multi-agents. Les thèmes concernés sont les contraintes temps réel, le raisonnement symbolique, l'asynchronisme. Chaque couche possède une vision de l'environnement et un mécanisme d'activation qui lui sont propres. Les trois couches définies dans InteRRaP sont en interactions fortes. Les interactions entre les couches s'effectuent par envoi de messages. Ceux-ci correspondent soit à une demande de prise en charge d'une situation non localement solvable, soit à des demandes d'exécution de solutions partielles. Ce découpage permet de réduire la complexité du système en ne considérant pour chaque couche que la représentation du monde qui lui est pertinente. Par exemple les croyances associées à la couche coopérative ont un haut niveau d'abstraction alors que la couche réactive doit répondre rapidement aux sollicitations et donc manipuler des croyances simples. Cette séparation des comportements permet une

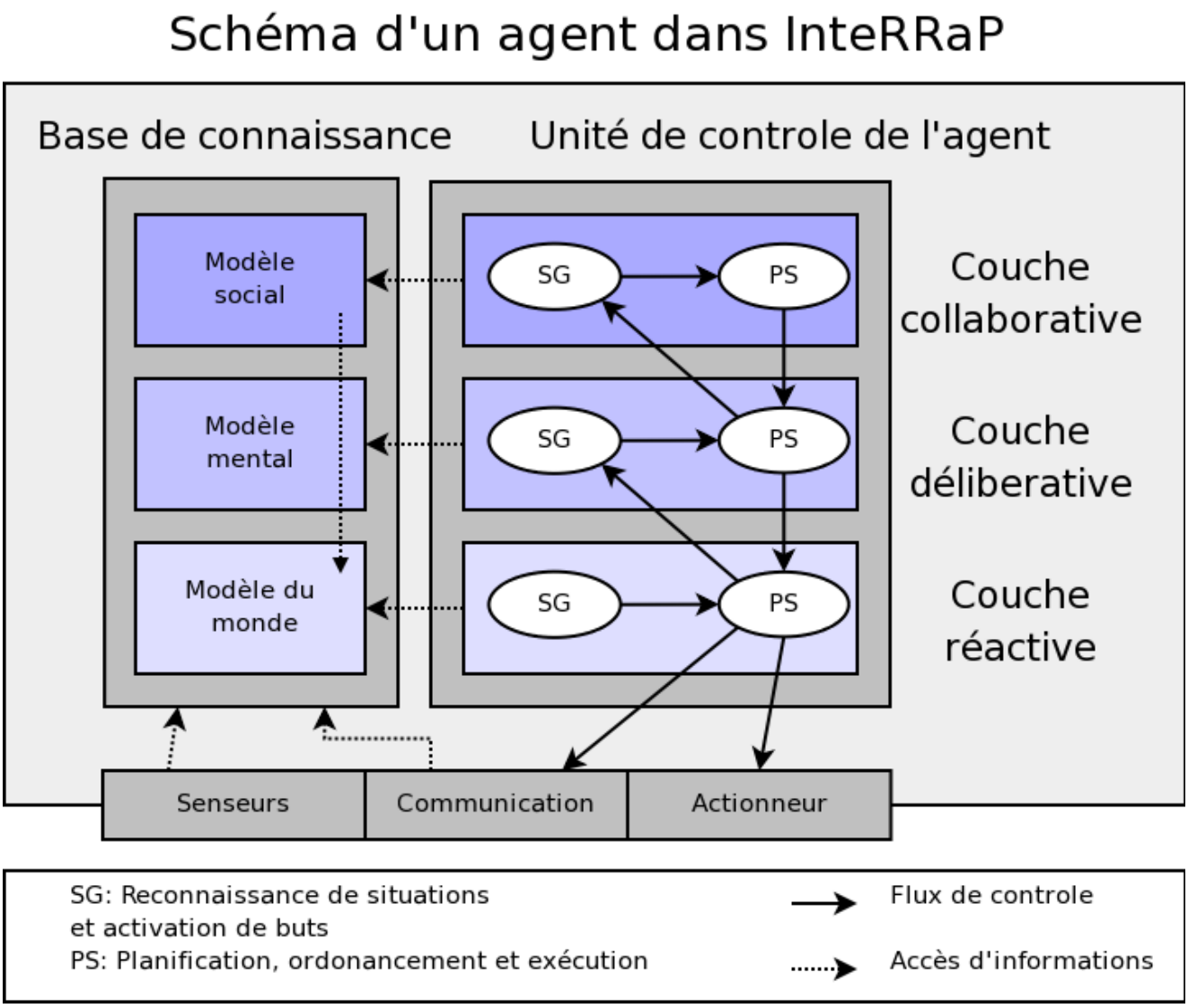

Fig. 1.3 – Aperçu schématique d'un agent dans InteRRaP

1. Mise à jour des croyances
2. Reconnaîssance des situations particulières
3. Génération des buts
4. Selection des opérations élémentaires
5. Planification des opérations à partir des buts
6. Exécution des opérations élémentaires

Fig. 1.4 – Le cycle de contrôle

approche incrémentale dans la reconnaissance des situations. C'est-à-dire que la reconnaissance des situations favorisera d'abord un comportement réactif, puis délibératif et enfin coopératif. InteRRaP repose sur une architecture BDI (Reliefs, Désires, Intentions) mise à jour à l'aide d'une AKB (Assertional Knowledge Base).

AKB : Assertional Knowledge Base Une base de connaissance assertionnelle sert pour le stockage des connaissances. Elle permet également de surveiller et de reconnaître des situations particulières. Une AKB fournit un ensemble structuré pour enregistrer et organiser les informations. Ainsi une AKB permet de définir des concepts dont les instances posséderont des attributs modifiables et des caractères non modifiables. Les attributs et les caractères sont typés et peuvent être ajoutés et supprimés. L'AKB va permettre de construire la connaissance que l'agent possède sur son environnement. Cette connaissance est composée de croyances qui sont des faits élémentaires. Ces croyances sont mises à jour à réglièrement de sorte que leur cohérence soit maintenue. On distingue la cohérence logique de la cohérence sémantique. La première assure que la négation d'une nouvelle croyance n'appartienne pas à L'AKB. La cohérence sémantique doit être explicité à l'aide d'axiomes par le concepteur du système. Exemple d'axiome : un objet ne peut pas être à deux endroits à la fois.

Le modèle de connaissance de l'agent dans InteRRaP est divisé en trois couches :

- Le modèle du monde : il contient toutes les croyances issues des senseurs
- le modèle mental : il regroupe l'ensemble des buts et les intentions de l'agent
- le modèle social : il regroupe les croyances sur les buts que poursuivent les autres agents

1.4.2 Couche réactive

Avant de voir comment ces différents mécanismes sont implantés dans la couche réactive il faut en définir le rôle. La couche réactive se charge de toutes les interactions avec le monde extérieur à l'agent. Elle permet de contrôler les actionneurs, d'acquérir des informations en provenance des senseurs et de communiquer avec les autres agents.

Modèle du monde InteRRaP fait l'hypothèse du symbolisme : C'est à dire que les interactions avec le monde extérieur se font au travers d'un formalisme symbolique qui repose notamment sur une base de croyances. Ainsi le premier rôle de la couche réactive est de répondre immédiatement à des évènements imprévus. Le second rôle est d'offrir une interface d'interaction avec le monde de "haut" niveau pour les couches supérieures. Ces deux rôles sont effectués à l'aide d'un même mécanisme : Le plan d'action élémentaire aussi appelé POB, pour Pattern Of Behavior. Un POB représente une séquence d'instructions "scriptés" ; comme par exemple : Reposer la caisse courante.

Reconnaissance de situations La reconnaissance de situations s'effectue au travers de conditions d'exécution de POB. Ceci est effectué à l'aide d'un service de requête active de L'AKB.

Grâce à l'hypothèse du symbolisme, il est facile de simuler le fonctionnement d'une AKB. Cependant sa mise en œuvre réelle peut présenter quelques difficultés : il faut acquérir les différents signaux (des capteurs) et puis les interpréter avec cohérence. Toute cette partie n'est pas traitée par InteRRaP.

1.4.3 Couche délibérative

Cette couche délibérative d'InteRRaP est associée aux mécanismes cognitifs. La couche délibérative regroupe les facultés cognitives de l'agent, or InteRRaP repose sur un modèle BDI. Le problème que doit donc résoudre cette couche est le suivant : *quel plan adopter pour satisfaire un but donné ?* Il existe deux approches pour la planification de tâches : (1) les approches dites " bottom up " tel que les MDPs (ou Processus Décisionnels de Markov) et les solutions " métiers " comme les systèmes experts et les bibliothèques de plans. Ce sont ces dernières qui sont utilisées dans InteRRaP, notamment à cause de leur simplicité.

L'émergence des situations est assurée grâce aux interactions avec les autres couches. Cette couche ne "reconnaît" les situations qu'à travers les activations de la couche réactive (qui correspondent à des évènements que la couche réactive n'a pas su résoudre) et les requêtes d'engagements (de mise en œuvre de plan collectif) de la couche collaborative. Il n'y a donc pas vraiment de reconnaissance de situation. L'objectif de cette couche est plutôt de trouver le plan qui permettra la résolution d'un problème donné. Enfin le comportement général de l'agent pourra être enregistré sous forme d'un but ; par exemple : décharger tous les camions. La principale difficulté réside dans l'implantation d'un planificateur de tâches : Cette difficulté sera résolue par une architecture modulaire qui se décompose en six interfaces :

Contrôleur : exécute le cycle de contrôle
Générateur : Détermine les plans possibles
Interpréteur de plans : évalue les plans
Évaluateur de plan : calcule une fonction d'utilité du plan
Ordonnanceur : planifie l'exécution des plans
Exécution : exécute et surveille le bon déroulement des plans

Remarque : Les buts ne vont servir que de clef d'indexation des plans ; c'est à dire qu'un plan est associé aux différents résultats auxquels il conduit. Deux buts sont déclarés similaires lorsqu'il existe une substitution pour passer de l'un à l'autre.

Bibliothèque de plans Les plans sont enregistrés dans une bibliothèque. Ils correspondent à des solutions " métier ". Ils peuvent mettre en œuvre soit des POBs (alors exécutés par la couche réactive) soit d'autres plans. Pour bien faire la différence entre un POB procédural et un plan, il convient de répondre à la question : Existe-t-il plusieurs façons d'effectuer ce traitement ? Si la réponse est négative, alors le plan est en fait un POB. Afin de garantir la cohérence (absence

de récursion infinie) l'ensemble des plan doit pouvoir être représenté sous forme d'un graphe acyclique.

Voici un exemple : Le plan courir a pour conséquences : (1) de se déplacer rapidement, (2) d'augmenter la fatigue, et (3) de se maintenir en forme. Si l'agent recherche à se maintenir en forme, il va déterminer que courir y parvient et va évaluer le bénéfice/coût associé. D'autres plans sont aussi évalués. L'agent choisit alors celui qui minimise son critère d'évaluation.

Utilité d'un plan Pour déterminer quel plan exécuter, il faut calculer les utilités associées aux plans. Cette mesure repose sur les notions de gain et de coût. Les coûts sont calculés relativement aux coûts estimés des actions qu'ils mettent en jeu. Plusieurs interactions sont possibles entre les couches collaborative et délibérative afin de déterminer le coût d'un plan. Ainsi lors d'une négociation, la couche collaborative peut demander l'évaluation de coût d'un plan particulier, sans vouloir que celui-ci soit exécuté. Pour modifier le comportement d'un agent on peut modifier sa fonction de calcul des coûts et bénéfices.

1.4.4 Couche collaborative

La couche Collaborative d'InteRRaP concentre tous les mécanismes d'interactions entre les agents, cette couche repose sur la négociation.

Les interactions entre les agents qui partagent un même environnement nécessitent parfois d'être coordonnées ; dans le cas de conflit notamment. De même lorsqu'une tâche ne peut pas être résolue localement un mécanisme de collaboration est utilisé. Pour assurer cette coordination InteRRaP utilise la négociation. Cette négociation suppose que les agents sont capables de communiquer. D'autres hypothèses sont faites sur le comportement des agents :

- ils sont supposés " honnêtes " : c'est-à-dire qu'ils doivent respecter leurs engagements et prévenir tout manquement
- ils partagent si nécessaire leurs connaissances.

Cependant aucune hypothèse ne sera faite sur leur comportement a priori. On définit la négociation à l'aide des trois notions suivantes :

- L'ensemble des solutions qui satisfont les buts des agents.
- Le protocole de négociation qui regroupe l'ensemble des interactions que peuvent avoir les agents pour converger vers une solution acceptée de tous.
- La stratégie qu'un agent va mettre en œuvre pour exploiter à ses fins le protocole de négociation.

Voici les étapes d'une collaboration et comment les trois notions précédentes sont utilisées :

1. Échange des états et des buts des agents en vue de la négociation
2. Sélection d'un problème à résoudre, si aucun problème solvable n'est trouvé la négociation s'arrête.

3. Choix des rôles des agents et du protocole de négociation. Les protocoles de négociation sont créés en fonction des situations.
4. Élection d'un chef qui calcule l'espace des solutions. Cet espace est basé sur des connaissances communes échangées par les agents. Ces solutions sont des plans collectifs.
5. Envoi de toutes les conditions de la négociation aux agents concernés.
6. Déroulement de la négociation en tour par tour.
7. Fin de la négociation et réalisation individuelle mais synchronisée des plans consentis.

Les plans collectifs mettent en œuvre un ensemble de plans individuels. Ainsi l'ensemble des solutions d'un problème peut vite devenir exponentiel. Ce calcul est donc simplifié.

1.5 Proposition

La programmation par aspect est un paradigme de programmation qui permet d'isoler et de composer des *modules* qui ont un même sujet de préoccupation. Ce paradigme de programmation est présenté plus loin. À partir de la programmation par aspect, il peut être possible de proposer un modèle d'agent dont le niveau d'hybridation est ajustable par l'utilisateur. La démarche est la suivante : À partir d'un modèle d'agent hybride, regrouper tout ce qui concerne son niveau d'hybridation et l'isoler, pour enfin en proposer un paramétrage.

Chapitre 2

InteRRaP à base de composants

Nous allons détailler les principes de base de la programmation par composants en détaillant le modèle de composant Fractal. Ensuite nous discutreons de l'implantation d'InteRRaP en composants.

2.1 Fractal

L'objectif de Fractal[1] est de proposer un modèle de programmation général (démarche et protocole) pour uniformiser l'implantation, le déploiement et la gestion (c'est à dire l'observation et la configuration) d'un système d'information.

2.1.1 Aperçu général

Fractal est un modèle de programmation par composants. Il respecte notamment les principes suivants :

- séparation des interfaces et des implantations
- programmation par composants
- l'inversion du contrôle des composants

Fractal est souple et n'impose aucune contrainte stricte. Plus précisément il n'impose pas une modélisation particulière, (mais un cadre dans laquelle l'effectuer et ce dont il a besoin) ni le paradigme objet (seulement une organisation des composants).

2.1.2 Les composants

L'idée directrice à Fractal est de considérer une application au travers des interactions entre composants. Ces composants interagissent uniquement au moyen d'interfaces. Ces interfaces décrivent les fonctionnalités d'un composant (ce qu'il est censé effectuer) ainsi que les propriétés non fonctionnelles (comment il se comporte). Fractal propose une organisation hiérarchique des

[1] Voici la page officielle de Fractal : http ://fractal.objectweb.org/.

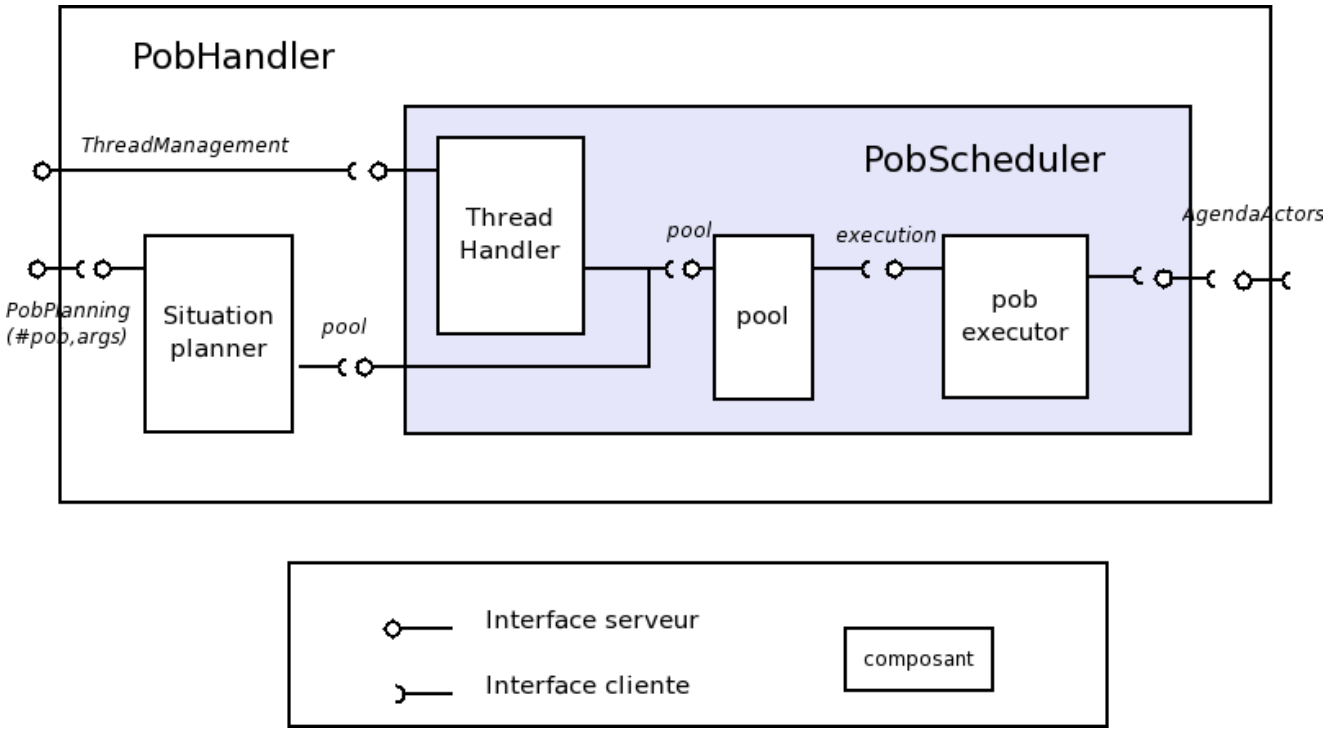

Fig. 2.1 – Exemple de 4 composants primitifs et de 2 composants composites. Seules les interfaces fonctionnelles sont représentées.

composants (qui peuvent en contenir d'autres). Des interfaces standards sont définies pour la manipulation de ces composants. En effet, comme les foncitonalités sont encapsulées dans des composants, ceux-ci doivent être "reliés" et "assemblés" pour fonctionner correctement. La figure 2.1 illustre cette organisation.

2.1.3 Les interfaces fonctionnelles

Chaque composant possède des interfaces serveurs qui correspondent aux services qu'il fournit. De même un composant définit les interfaces clientes qui représentent ses dépendances (les services requis) vis-à-vis d'autres composants. Les interfaces clientes et serveurs sont propres à un type de composant, elles doivent donc être explicitement créées lors de la création d'un composant.

2.1.4 L'inversion de contrôle des composants

Le Fractal repose sur l'inversion de contrôle des composants. Lorsque chaque composant exprime ses besoins (avec les interfaces clientes) il n'est plus nécessaire pour aucun composant de manipuler directement un autre composant. Ceci s'effectuera "depuis l'extérieur" du composant en question. Cela permet notamment de reconfigurer un ensemble de composants en changeant leurs associations client/serveur. Par exemple soit A, B et C trois composants tels que A est client

de B pour l'interface i. Si C possède également une interface serveur de type i, il est alors possible de changer l'association client/serveur entre A et B pour que C devienne le serveur de A.

2.1.5 Les interfaces non fonctionnelles

Les interfaces non fonctionnelles (ou interfaces de contrôles) permettent de manipuler les composants. Dans Fractal tous les composants possèdent au moins une interface non foncitonnelle : l'interface composant. Cette interface renseigne (permet d'accéder à) toutes les autres interfaces d'un composant. Une interface est définie pour chaque opération. Par exemple, une interface de mise en association (BindingController) permet de définir les relations client/serveur. De plus les composants peuvent être encapsulés dans d'autres composants (appelés composants composites). Ainsi les composites possèdent une interface d'ajout et de suppression des "sous-composants". En fait les interfaces non fonctionnelles peuvent décrire tout type d'opération agissant sur le comportement d'un composant.

2.1.6 Factory, template et bootstrap

Pour créer un composant, il est nécessaire de faire appel à un composant usine (ou *Factory* en anglais). Ce composant se charge de tous les détails de la création d'un composant : interfaces, implantation ou sous-composants éventuels. Cependant une usine est elle-même un composant et nécessite aussi d'être créée par une autre usine. C'est pourquoi chaque implantation de Fractal propose un composant spécial appelé *bootstrap* et à partir duquel il est possible de construire les divers types et usines nécessaires. Un autre mécanisme existe : Le patron (ou *template* en anglais). Son fonctionnement est similaire à l'usine sauf qu'il est lui-même un composant du type souhaité ayant la capacité de se dupliquer.

2.1.7 Récapitulatif

Voici une liste des fonctionalités que Fractal propose :

- Des interactions entre les composants s'effectuent par le biais d'interfaces clientes et serveurs spécifiées en IDL (Interface Description Language).
- Les interfaces d'un composants sont de deux natures : fonctionnelle et de contrôle. On y accède au travers de l'interface CompomentInterface.
- Les attributs définissent l'état (liés au fonctionnement) du composant (ex : couleur, taille maximale du cache, nom). On leur associe une interface de contrôle (AttributeController : setName, g
- On manipule les interfaces clientes au travers d'un contrôleur (BindingController). Ceci permet l'inversion de contrôle des composants.
- Une encapsulation hiérarchique des composants. Celle-ci se manipule au travers de l'interface ContentController.

– Un contrôle d'activité des composants (LifeCycleController) permet d'utiliser notamment les interfaces BindingController et ContentController sans risquer de corrompre l'activité du composant.
– Les composants sont créés à l'aide de composants de type Factory ou Template.

Tous ces dispositifs permettent de créer, d'organiser et de modifier des composants ainsi que leurs interactions.

2.1.8 Conclusion sur Fractal

Ainsi Fractal propose une démarche unifiée de conception de système d'information. Cette méthode ne limite pas le concepteur dans un "design" d'application, mais elle unifie plusieurs thématiques du génie logiciel (réutilisation, déploiement, (re)configuration...) et propose comme solution générale un formalisme particulier qui permet clairement de modifier la structure et le comportement de l'application (notamment grâce à l'encapsulation et au contrôle inverse).

2.2 Implantation

2.2.1 Conception en composants

Afin de réaliser le modèle d'agent InteRRaP à l'aide de composants, nous avons souhaité conserver autant que possible la structure logique de l'agent telle qu'elle est décrite dans le livre (Ml94). La figure 2.2 illustre comment l'agent a été modélisé.

À cet effet les POBs et les plan sont représentés par des composants. L'AKB est aussi modélisée par un composant, cependant les connaissances sont de simples objets. Dans la figure 2.2 on peut constater les similitudes structurelles entre les 3 couches. Enfin, on peut noter que seule la couche réactive entre en contact avec l'AKB et l'interface avec le monde (les composants Middleware et Agenda). C'est en effet un point laissé en suspens dans l'ouvrage (Ml94). Il y est écrit que seule la couche réactive communique avec l'interface au monde (comopsée des senseurs et des actionneurs). Mais on pet également y lire que chaque couche receptionne les messages qui lui sont destinés pour les traiter. D'autre part, l'AKB est incapable d'inférence et doit fournir une "vision" du monde adaptée à chaque couche. Malheureusment une telle AKB est complèxe à mettre en œuvre et tout comme Mr Jörg Muller nous n'avons pas réalisé une telle base des connaissances. Nous avons choisi un seul méchanime pour résoudre ces deux problèmes : les POBs réatifs. En effet les POBs réactifs, sont par nature des lieux de traitement permanents. Il vont donc nous permettre :

– De transmettre les messages aux couches concernés.
– D'abstraire les connaissances pour les couches suppérisupérieures

L'architecture de la couche réactive est représenté figure 2.3. Comme nous venons de voir, cette couche à un rôle primordiale dans le fonctionnement des autres couche. Cet eentrelacementdes responsabilité sera étudié dans le chapitre suivant.

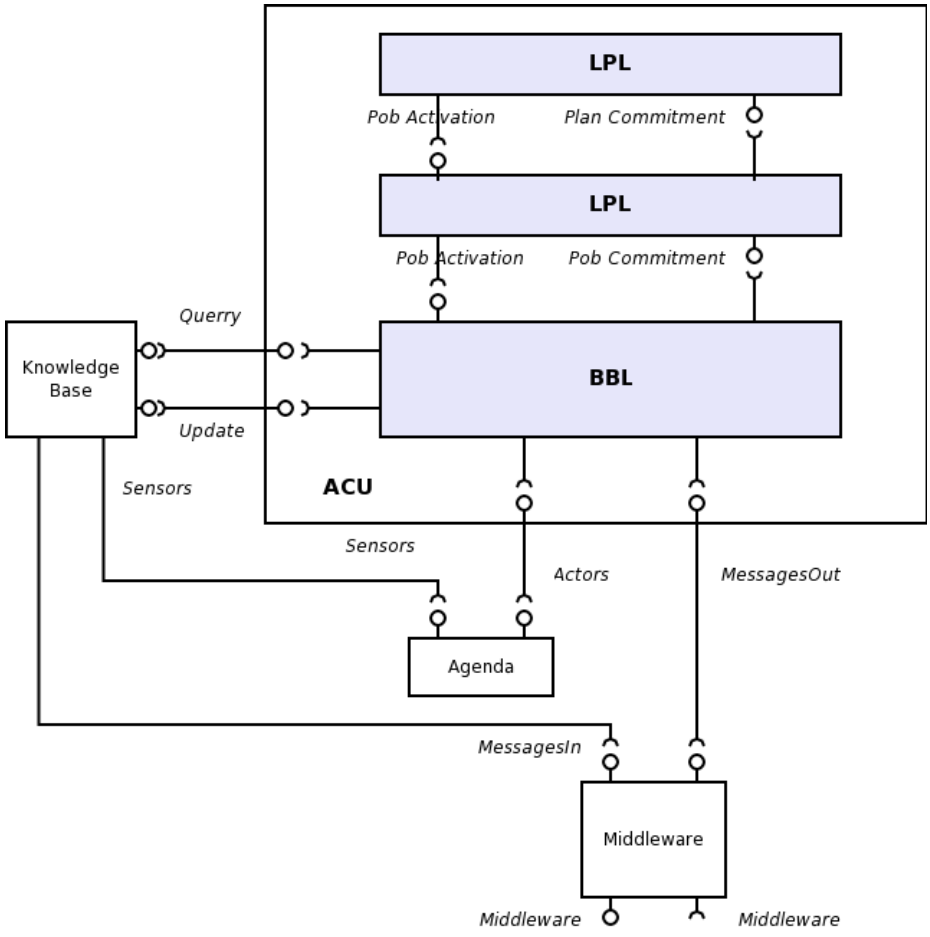

Fig. 2.2 – Apperçu des composants d'InteRRaP.

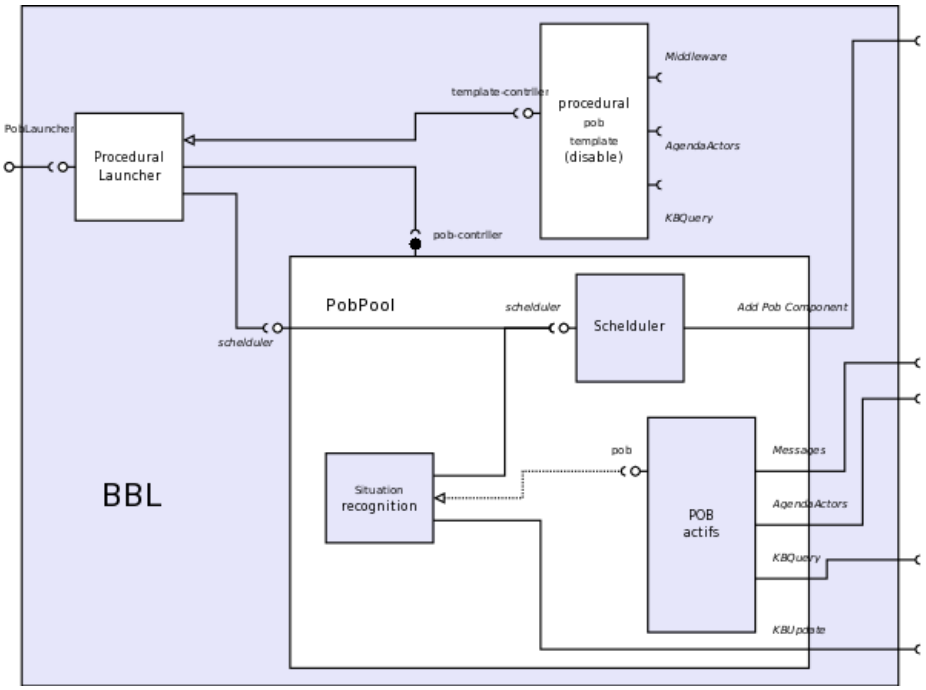

Fig. 2.3 – Détail des composants de la couche réactive

2.2.2 Difficultés d'implantation

J'ai rencontré des difficultés lors de l'implantation d'InteRRaP en composants. Cette partie tente de les analyser afin de proposer une critique constructive du modèle de composants Fractal. Le point délicat concerne la dynamicité de l'architecture des composants. La dynamicité regroupe trois phénomènes : (1) la création et l'insertion de nouveaux composants durant l'exécution ; (2) la modification des liaisons d'un composant et (3) la migration d'un composant d'un composite à un autre. Fractal propose des interfaces de ccontrôles afin de supporter toutes ces opérations. Cependant il n'est pas aisé de déterminer la bonne architecture qui sera effectuer ces opérations dans le respect des règles de Fractal. Par exemple l'interface bindingController permet de manipuler les liaisons internes d'un composite. Ces modifications sont donc seulement accessibles depuis l'extérieur et ne pourront pas être entreprises "seules" depuis l'intérieur d'un composite. Afin de modifier son environnement un composant a donc besoin d'une "aide extérieure". Or recourir à un autre composant extérieur nécessite de modifier le composite lui-même (pour qu'il déclare cette interface requise). En résumé, pour qu'un composant procède dans son exécution à des modifications sur l'architecture il est souvent nécessaire de prévoir ces modification pendant la conception. Si cette nécessité n'est pas étudiée auparavant l'introduction d'un comportement dynamique peut introduire de modifier la structure hiérarchie des composants (i.e. les composites). Deux situations qui permettent d'illustrer plus clairement cette idée sont détaillées dans les annexes.

Chapitre 3

InteRRaP à base d'aspects

3.1 La programmation par aspect

3.1.1 Présentation

La programmation par aspect est un paradigme de programmation qui permet de découpler les fonctionnalités d'un logiciel qui n'ont pas les mêmes préoccupations. Le principe est d'isoler les propriétés transverses (aspects) d'une application dans des modules. Ces aspects sont ensuite "tissés" pour former l'application finale. La figure 3.1 schématise ce principe.

Composition d'aspects On peut schématiser l'exécution d'un programme informatique par un couple <programme, interprète >. Ce modèle est tout à fait général et l'interpréteur peut-être de très bas niveau (comme un processeur) ou bien de niveau plus abstrait (comme une machine virtuelle Smalltalk). Dès lors il existe deux approches pour composer les aspects :

- par transformation de programme (ex : AspectJ)
- par transformation d'interprète (ex : Fractal-AOP voir (Pes04))

Transformer l'interprète, peut être complexe et nécessite un cadre. Les langages dynamiques offrent naturellement ce cadre de transformation à travers la *réflexivité*. Nous allons maintenant présenter *MetaclassTalk*, une plateforme de programmation par aspect.

3.1.2 MetaclassTalk : une plateforme unifiée

La programmation par aspect repose sur la séparation des préocupations. Cela signifie que les fonctionalités sont regroupées par sujet. Si cette séparation est appellée *crosscutting* en anglais (ce qui se traduit littéralement par "découpage en croix") elle n'est pas la réelle problématique de la programmation par aspect. Celle-ci doit proposer un moyen de programmer séparément les aspects et de les tisser pour obtenir finalement le comportement mixte souhaité. Afin de programmer ces aspects MetaclassTalk propose deux méchanismes :

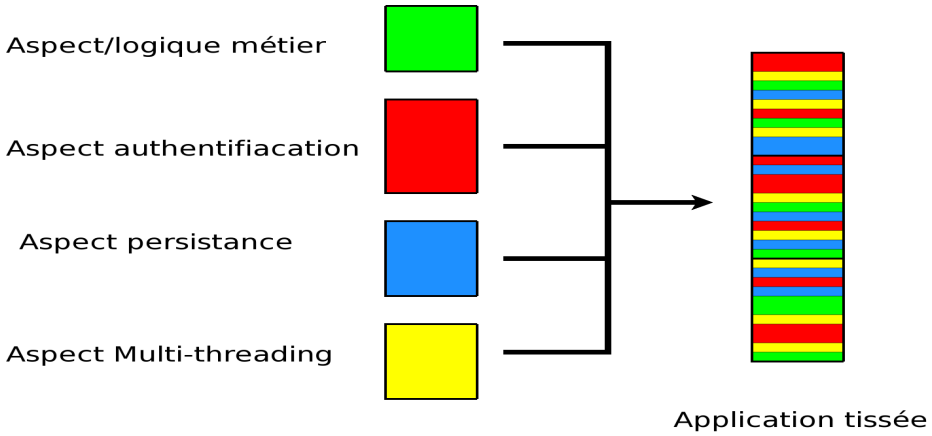

Fig. 3.1 – Illustration symbolique du paradigme aspect

- Le découpage *dynamique*. Il permet l'introduction de traitements en certains points du flot d'exécution. Ces points sont appellés *points de coupes* (pointcuts en anglais). Par exemple l'appel d'une methode est un point de coupe potentiel. Ce point de coupe va permettre l'introduction de traitement dans le flot d'exécution. Ce méchanisme est implanté dans MetaclassTalk à l'aide de la réflexivité : c'est le méta-objet qui va supporter les modifications.
- Le découpage *statique*. Il permet de modifier la structure même des entités. Ainsi il est possible d'ajouter des variables d'instances et des méthodes à un objet. Il y a modification structurelle des objets. On appelle *mixin* un ensemble de fonctions et d'instances de variables qui pourra être partagé entre plusieurs classes quelconques. Les mixins et l'héritage sont deux moyens de réutilisation de code.

3.2 Hybridation de l'agent

Afin de permettre l'ajustement de l'hybridation, il convient d'isoler dans InteRRaP tout ce qui concerne l'hybridation comme définie au chapitre 1. Nous avons donc recherché à mettre en évidance tout les méchanismes qui, dans InteRRaP, participent à l'activité d'une couche particulière dans le processus de décision. Trois points ont relevé notre attention ; il s'agit de fonctionnalités qui concernent :

- *Le passage d'une couche à l'autre* : Lorsqu'un échec est rencontré par une couche, celle-ci envoie une requête d'activation pour que la couche supérieure traite cet évènement
- *La granularité des actions* : Afin de permettre à l'agent d'être plus ou moins réactif, il est possible de d'ajuster le pas d'exécution des plans et des POBs.
- *La parallélisation des traitements* : Pouvoir répartir la charge selon les couches permet également de modifier l'hybridation de l'agent. Ceci peut être effectué en allouant un pool de threads pour chaque couche.

Nous allons détailler ces trois pistes dans les sections suivantes. Afin d'illustrer nos propos, nous nous appuierons sur le scénario décri dans l'annexe D qui décrit une situation dans laquelle des agents sont associés à des agendas.

3.2.1 Gestion des échecs

La programmation d'un agent doit considérer que toutes les actions exécutées sont susceptibles d'échouer. Se pose alors la question de la détection et du suivi de ces échecs. À ce sujet InteRRaP propose un système complet de gestion des échecs.

Les POBs Les POBs possèdent un mécanisme sophistiqué d'exécution et de détection des échecs. Pour rappel, le cycle de vie d'un POB est le suivant : initialisation puis exécution. L'exécution consiste en un cycle dont voici les étapes : exécution d'une étape élémentaire (step) ; teste des conditions de succès et d'échec et traitement adéquat. Ainsi les échecs sont "dynamiquement" détectés et traités immédiatement. Plusieurs traitements d'échecs sont envisageables :

- Demande d'activation vers la couche délibérative. Elle aura la charge de résoudre le problème. Cette solution délègue la responsabilité à la couche supérieure. Son bon fonctionnement dépend donc des compétences de la couche suppérieure.
- Attente d'un délai aléatoire puis ré-initialisation du POB. Cette démarche fonctionne pour le cas des conflits de ressources entre les agents. Mais elle est inutile lorsque le problème n'inclut pas d'aspect temporel (ex : impossible de trouver une date disponible dans un agenda complet).
- Recherche d'un autre POB qui satisfait le même but. Comme nous l'avons vu il peut exister plusieurs versions d'un même POB. En cas d'échec, dans certains cas, un POB de substitution peut donc être exécuté.

Ces trois "stratégies" sont en fait complémentaires. Cependant elles participent à la gestion des ressources dans la mesure où un choix d'action est effectué.

Les plans Un but est satisfait de la manière suivante :

- choxi du plan : parmit l'ensemble des plans qui satisfont ce but, un seul est retenu selon une fonction d'évaluation.
- exécution et interprétation paresseuse : s'il fait intervenir de nouveaux buts ceux-ci sont résolus au moment de l'exécution. s'il fait intervenir des POBs alors ceux-ci sont exécutés par la couche réactive.

Deux types d'échecs peuvent intervenir : Soit aucun plan ne satisfait le but considéré, soit le POB lancé a échoué. Les échecs sont traités de la manière suivante :

1. Si d'autres plans qui satisfont le but existent, ils sont exécutés successivement.
2. Une demande de prise en charge est envoyée à la couche supérieure.

Dans cette couche il n'y a donc pas d'influence sur les ressources dans la gestion des échecs.

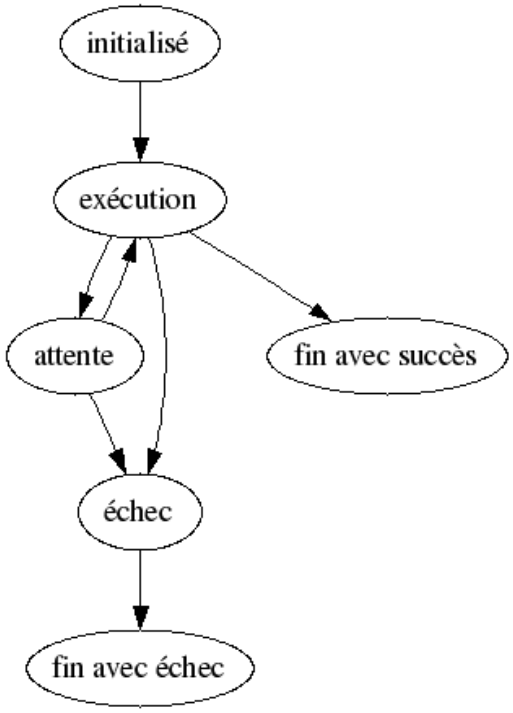

Fig. 3.2 – Les transitions possibles entre les différents états d'un script

3.2.2 Implantation des POBs et des plans

Ici le terme script désigne indistinctement les POBs et les plans. Dans InteRRaP, les capacités de l'agent sont enregistrées sous forme de séquence d'actions (scripts). On distingue deux types de scripts : les POBs, qui correspondent à des compétences précises et hors contexte ; et les plans qui sont des instructions plus complexes dont il peut exister plusieurs versions concurrentes. Les plans satisfont des buts particuliers.

Exécution des scripts Les POBs et les plan sont exécutés de la même manière : ce sont des scripts. Nous allons décrire le processus d'exécution des scripts. Les scripts ont un cycle de vie qui correspond aux différents états auxquels ils peuvent accéder. Ces états sont schématisés par la figure 3.2. La transition entre ces états est assurée en partie par le Scheduler et l'Interpreter.

Les POBs Les POBs sont des comportements élémentaires. Ils représentent des briques (actions élémentaires) qui peuvent être composées pour obtenir des comportements plus complexes. Ils participent donc à l'hybridation. On distingue deux types de POBs, les POBs réactifs (qui sont activés par une condition) et les POBs procéduraux qui sont explicitement lancés. Les dépendances entre les POBs sont illustrés figure 3.3. Les POBs réactifs ont plusieurs usages :

– Ils servent à la communication entre les couches délibératives des agents. En ce sens ils participent à l'activité "délibérative" de l'agent. On peut citer à titre d'exemple les POBs

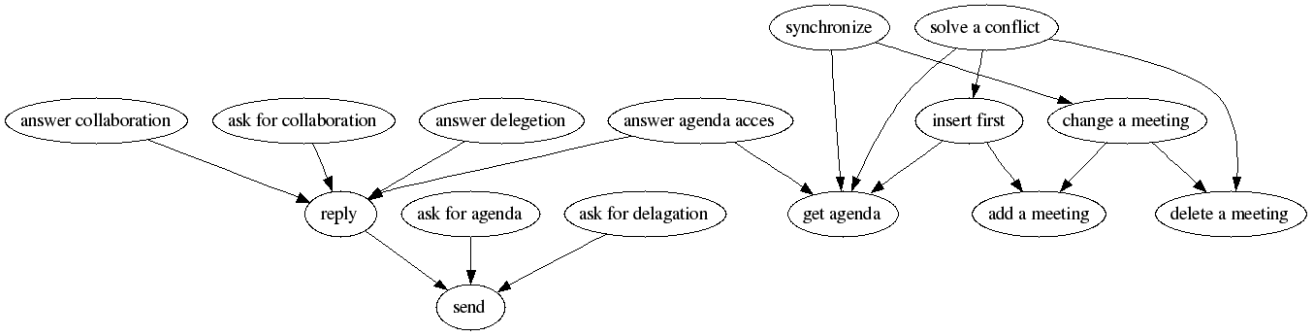

Fig. 3.3 – Réseau des dépendances entre les POBs pour l'application des agendas

suivants : answerCollaboration et answerDelegation.
- Ils permettent d'ajuster directement le comportement de l'agent. Par exemple ils peuvent surveiller l'apparition d'anomalies (agenda incohérent dans notre exemple) et exécuter les traitements appropriés.
- Ils forment le comportement "par défaut" de l'agent, c'est-à-dire quand celui-ci n'est pas solicité. Par exemple, un agent peut synchroniser périodiquement sont agenda.

On retiendra qu'un certain nombre de POBs réactifs produisent naturellement de l'activité, et cette activité consomme des ressources. Les POBs concernś sont les suivants :
- answerCollaboration
- answerDelegation
- synchronize
- solveConflict

Ainsi tout les agents doivent posséder ces POBs pour fonctionner correctement mais ils en posséderont des versions différentes. Concrètement pour un agent plutôt réactif, les traitement associés à ces POBs s'effectuerons à l'aide d'autres POBs alors que des agents délibératifs et collaboratifs auront des implantations différentes : ce seront des demandes d'activations de la couche délibérative.

Les plans Les plans satisfonts des buts et ils peut exister plusieurs plans qui satisfont un même but. Dans ce cas il faut alors évaluer les plans pour sélectionner le plus approprié. Cette sélection peut s'effectuer selon plusieurs critères comme la fiabilité, les ressources nécessaires et disponibles ou encore le temps d'exécution. Ainsi la manière d'évaluer les plans va conditionner le comportement de l'agent et son usage des ressources.

3.2.3 Granularité de l'exécution

Pour la couche réactive, Il est par exemple possible de modifier l'exécution des POBs (la méthode step) pour modifier la durée du cycle : update/execution/test afin d'introduire plus

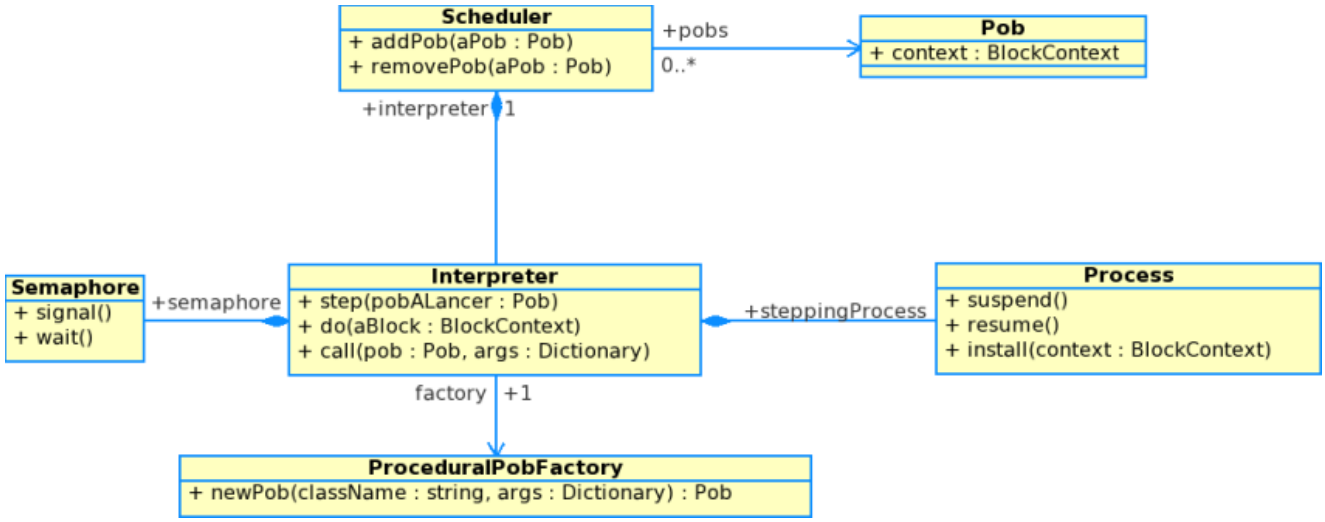

Fig. 3.4 – Diagramme de classe de l'interprétation de POBs

ou moins de réactivité. Cette exécution "pas à pas" est obtenue en exprimant les actions des POBs à l'aide d'un "langage" rudimentaire (Comme l'a suggéré Serge !). Ce langage procÃl'dural permet de suspendre l'exécution des POBs à certains points précis. Ainsi l'exécution des POBs est ordonnancée grâce au PobScheduler, et interprétée par un PobInterpreter. Celui-ci se charge de l'exécution proprement dite de POBs. Les figures 3.4 et 3.5 illustrent ces comportements.

Afin de pouvoir suivre le déroulement des POBs ils sont interprétés "pas à pas". Le cycle d'exécution des POBs effectue selon la granularité d'exécution des POBs. Cette granularité est dispersée dans la classe PobInterpreter et la programmation par aspects nous permettra de regrouper cette fonctionnalité. Cette granularité est définie dans les méthodes suivantes :

- PobInterpreter>>done
- PobInterpreter>>step:
- PobInterpreter>>call:withArgs:
- PobInterpreter>>terminate:

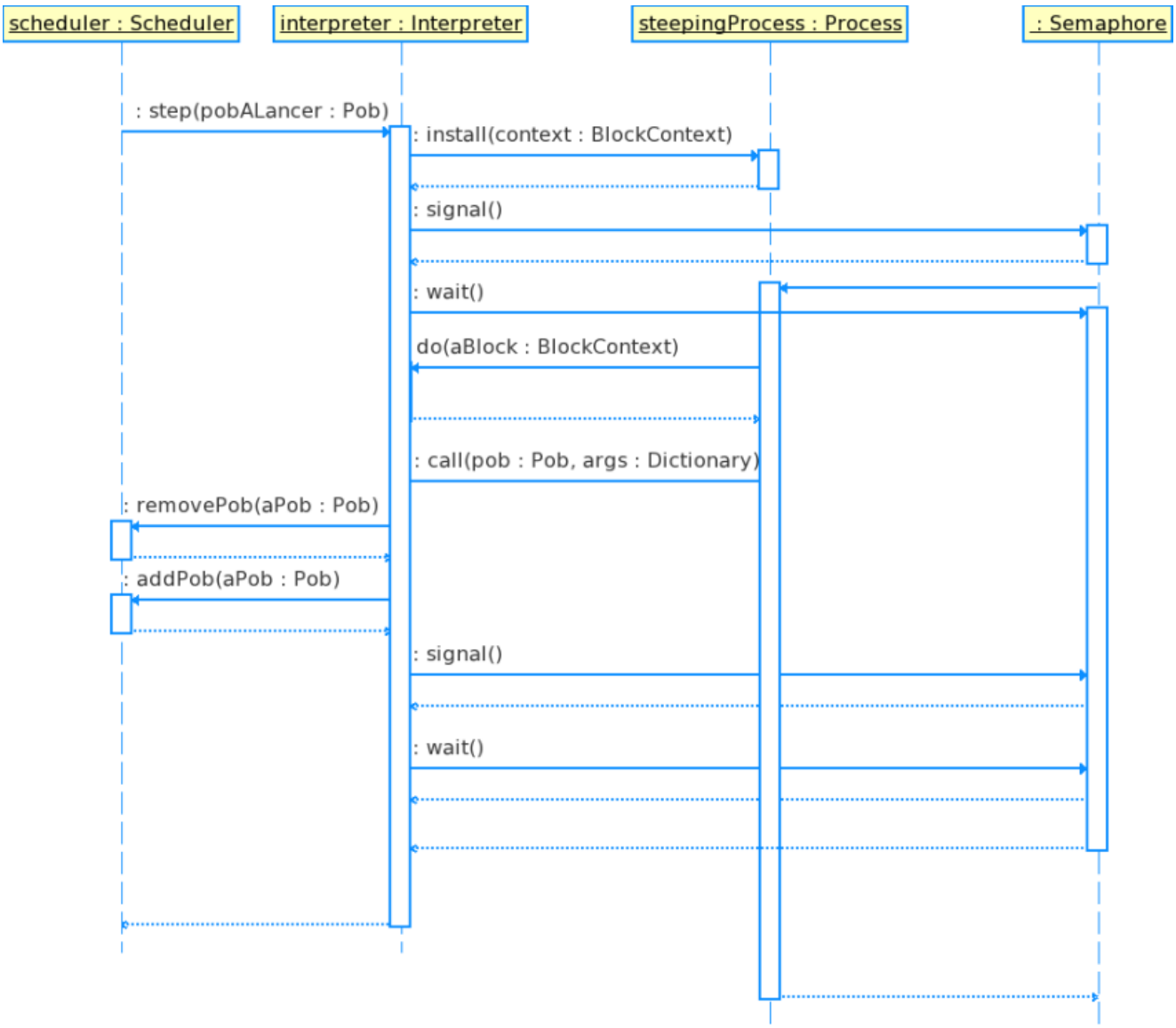

Fig. 3.5 – Diagramme de séquence : Exemple de lancement d'un POBs par un autre (PobInterpreter>>call:withArgs:)

Chapitre 4

Conclusion et Perspectives

4.1 Bilan du travail effectué

J'ai étudié en détail l'architecture et le fonctionnement d'InteRRaP. Je me suis ensuite formé à la programmation par composants en utilisant le framework FracTalk, une version de Fractal implantée en Smalltalk. J'ai travaillÃl' à l'implantation d'InteRRaP sous forme de composants. Nous avons également imaginé un scénario de mise en œuvre d'agents InteRRaP qui illustre la problématique des ressources (voir annexe D). D'autre part nous avons recherché toutes les fonctions d'InteRRaP qui permettent d'envisager l'hybridation de ce modèle d'agent comme un aspect.

4.2 Travail en cours

Face aux difficultés rencontrées à exprimer InteRRaP sous forme de composants, j'ai implanté une autre version en Smalltalk. Celle-ci ne comporte que les couches réactives et délibératives. Le travail sur l'aspect et l'hybridation reste à être validé techniquement même si les pistes exploitées semblent prometteuses. Enfin l'application des agendas n'a donc pas pu être testée.

4.3 Conclusion

Peu de recherche a pour le moment été effectuée en conception d'agent d'un point de vue génie logiciel. Tout semble indiquer que ces deux domaines convergent (voir (Ode04)). En effet un agent n'est rien d'autre qu'un programme ayant certaines caractéristiques, tout comme un programme est un agent un peu simple. Ainsi les thématiques des agents (comme l'autonomie et l'adaptation) trouvent des réponses originales dans le génie logiciel (comme la migration de composants ou bien les interfaces optionnelles). Réciproquement les agents ont influencé le génie logiciel pour y introduire les notions de rôle et de groupe. Génie logiciel et systÃĺmes multi-agents gagnent à se rencontrer.

Concernant notre étude, elle reste en cours de réalisation. Nous avons cependant montré qu'il existe dans InteRRaP des foncitonnalités transverses qui concernent la gestion des ressources lors du processus de décision. De telles techniques permettent de modifier de façon transparente une fonction essentielle (comme le processus de décision) de l'agent.

Annexe A

FracTalk, une implantation de Fractal en Smalltalk

Présentation FracTalk est un paquetage de l'environnement Squeak qui implémente Fractal. Le site officiel est disponible à l'addresse : http ://csl.ensm-douai.fr/FracTalk. Il est librement accessible via l'outil monticello dans squeak à l'adresse suivante :

```
MCHttpRepository
location: 'http://squeaksource.com/FracTalk'
user: ''
password: ''
```

A.1 Description d'un exemple

L'objectif de cette section est de présenter la démarche de réalisation d'une application Fractal avec FracTalk. Afin de présenter FracTalk, tout au long de cette partie nous allons développer un exemple. Il s'agit d'un petit système de communication, où plusieurs personnes (bientôt composants) souhaitent partager leurs idées ensemble.

A.2 Conception

Dans cet exemple, plusieurs entités souhaitent partager leurs idées. Nous allons mettre en œuvre un composant Media, qui se chargera de propager des messages à des *Speakers*. Les Speakers s'enregistrent sur le *Media*. Aussi ils peuvent "parler" et "entendre" tout ce que se disent les autres. Un diagramme UML correspondant est représenté figure A.1.

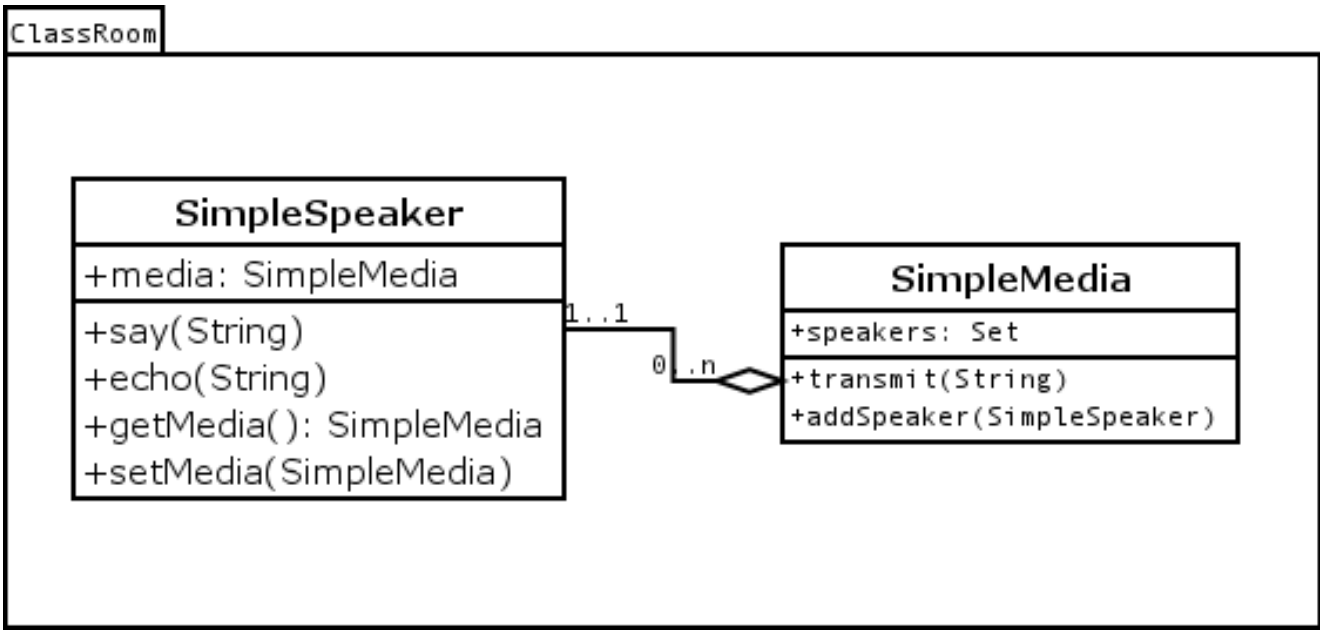

Fig. A.1 – Présentation de l'exemple

A.3 Déploiement dans FracTalk

Afin de mettre en œuvre ces objets nous allons leur associer des composants. L'application imaginée correspond à une salle de classe où des élèves peuvent entrer et sortir.

Composants : Nous allons modéliser les élèves par des composants (Speaker). De même le vecteur de communication (media) sera un composant (Media). Enfin la classe sera également un composant. Les types de composants seront donc les suivants :

- Classroom (Composant composite)
- Speaker (Composant primitif)
- Media (Composant primitif)

Définition des interfaces : Les composants Speaker et Media correspondant sont des composants dits "primitifs". Leur membrane permet d'accéder directement aux méthodes qui implémentent leurs services. En bref, ils sont représentés par des objets (appelés posos) qui contiennent l'implantation des services. Le composant Classroom est qualifié de composite car il enveloppe les autres composants. La figure A.2 représente l'oganisation des composants.

Définition des contrats - Interfaces fonctionnelles : Afin de pouvoir interagir de façon transparente, les services et les dépendances entre composants sont décrits au moyen d'interfaces. Les services fournis par le composant forment ses interfaces serveurs. Et les services qu'il requière pour fonctionner sont ses interfaces clientes. Pour que deux composants communiquent, il faut

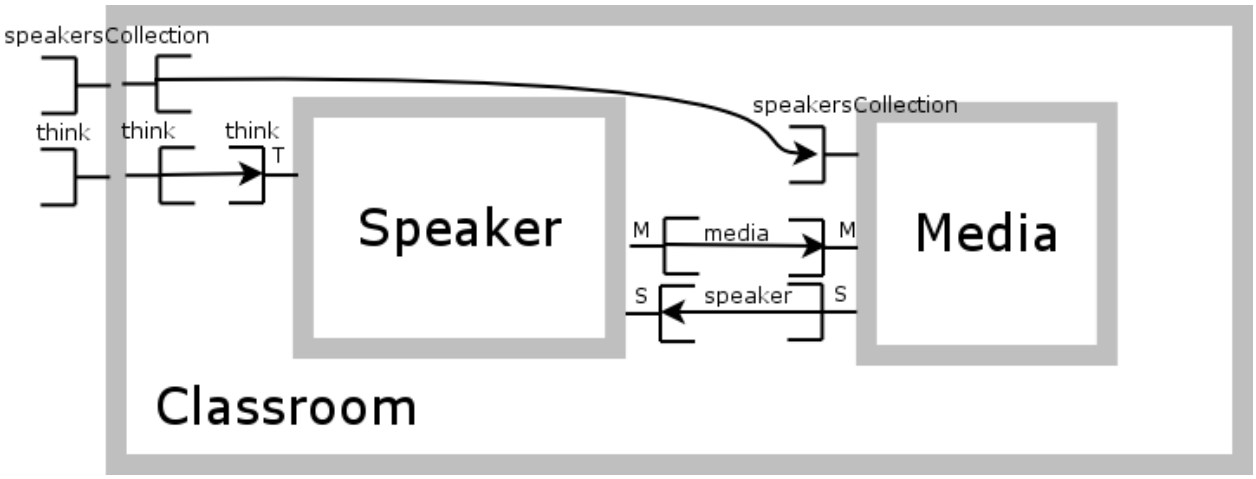

Fig. A.2 – Schéma des composants

que l'interface cliente soit incluse dans l'interface serveur. Par la suite nous les considèreront identiques, mais ceci n'est pas nécessaire. Ainsi nous définissont 4 types d'interfaces :

- L'interface media qui comprend une fonction transmit :
- L'interface speaker qui comprend une fonction echo :
- L'interface think qui comprend une fonction say :
- L'interface speakersCollection qui comprend une fonction addSpeaker :

Un speaker possède les interfaces speaker en rôle serveur et media en rôle client. Ses deux interfaces ont une cardinalité singleton. Un media possède aussi ces deux interfaces, mais avec des rôles inverses et une cardinalité multiple pour l'interface speaker. La figure A.2 illustre le modèle.

A.4 Implantation

Granularité Dans l'exemple que nous développons chaque composant primitif sera implémenté par une classe. Ceci n'est pas obligatoire, mais c'est souvent le cas. Il est important que chaque composant manipule les autres composants à l'aide des interfaces. Pour pouvoir utiliser le bindingController, il est nécessaire (avec FracTalk) de nommer un attribut avec le même nom que l'interface qui permet d'y accéder. Par exemple, dans le cas de l'interface media il faut que le champ de l'attribut de la classe SimpleSpeaker soit aussi media. Voici par exemple la methode say : de l'objet SimpleSpeaker :

```
say: aString
"Tell aString to the group through my media"
media
        ifNil: [Transcript cr; show: 'no media to dsplay: ', aString]
        ifNotNil: [(media getFcInterface: 'media') transmit: aString]
```

La gestion des attributs effectuée via l'AttributeController nécessite de définir pour chaque attribut un setter (setAttribut :) et un getter (getAttribut). Afin de permettre la manipulation de l'attribut media de Speaker, les methodes getMedia et setMedia : doivent être implantées dans le poso SimpleSpeaker. Cependant, un composant ne doit possèder que l'interface serveur qu'il utilise, et non le composant qui la fournit. Cela signifie concrètement que les implantations sont effectuées sans tenir compte des membranes. Le listing suivant (issue d'une classe de test) vérifie la relation de deux composants au travers d'un attribut :

```
self assert:
        ((speakerComponant getFcInterface: 'attribute-controller')
                getAttribute: 'media') = (mediaComponant getFcInterface: 'me
```

On peut y remarquer que le Speaker ne possède que l'interface media du composant Media et non le composant lui-même. Si le speaker doit utiliser plusieurs interfaces du Media, alors plusieurs attributs (nommées par les interfaces) seront nécessaires. Cette contrainte explique pourquoi l'attribut doit possèder le même nom que l'interface à laquelle il correspond.

A.5 Mise en service et configuration

Bootstrap Le processus de création d'un composant met en œuvre un composant spécial appelé bootstrap. Le composant bootstrap implante les interfaces TypeFactory et GenericFactory. A l'aide de ces interfaces il est possible de créer les types des interfaces, puis avec les posos on peut créer les composants primitifs, puis les composants composites. Une fois les composants créés, on peut les configurer à l'aide des interfaces de composition et de liaison.

Makers Afin de simplifier la mise en œuvre d'un ensemble de composants, FracTalk introduit la notion de makers. Le role d'un maker est de fournir immédiatement des composants particuliers et fonctionnels. Les makers regroupent tous les méchanismes de création précédement décrits. Un exemple de maker est en annexe.

Démarrage des composants Afin de pouvoir utiliser les services d'un composant il est nécessaire de l'activer. Ceci s'effectue via l'interface LyfeCycleInterface comme suit :

(media getFcInterface : 'lifecycle-controller') startFc

Surveillance et maintenance Grâce au méchanisme d'interfaces, il est possible d'introspecter les composants. On peut donc facilement les surveiller et les modifier.

Annexe B

Interface de programmation Fractal

- interface Component :

```
Object getFcInterface(String)
Object[] getFcInterfaces()
Type getFcType()
```

- interface Interface :

```
String getFcItfName()
Component getFcItfOnwer()
Type getFcItfType()
boolean isFcInternalItf()
```

- interface AttributeController : aucune (voir BindingController)
- interface BindingController :

```
void bindFc(String, Object)
String[] listFc()
Object lookupFc(String)
void unbindFc(String)
```

- interface ContentController :

```
void addFcSubcomponent( Component )
Object getFcInternalInterface( String )
Object[] getFcInternalInterfaces()
Componant[] getFcSubcomponents()
void removeFcSubcomponent(Component)
```

- interface NameController :

```
String getFcName()
void setFcName(String)
```

- interface LifeCycleController :

```
String getFcState()
void startFc()
void stopFc()
```

- interface SuperController :

```
Componant[] getFcSuperComponents()
```

- interface Factory :

```
Object getFcContentDesc()
Object getFcControllerDesc()
Type getFcInstanceType()
Component newFcInstance()
```

- interface GenericFactory

```
Component newFcInstance()
```

- interface InterfaceType :

```
String getFcItfName()
String getFcItfSignature()
boolean isFcClientItf()
boolean isFcCollectionItf()
boolean isFcOptionalItf()
```

- interface ComponentType :

```
Type getFcInterfaceType(String)
Type[] getFcInterfaceTypes()
```

- interface TypeFactory :

```
InterfaceType createFcItfType(String, String, boolean,
boolean, boolean)
ComponentType createFcType(InterfaceType[])
```

Annexe C

Fractal et la dynamicité

C.1 Premier exemple : Le motif stratégie

Nous allons illustrerl'usage du schéma de conception “stratégie” avec Fractal. Le motif stratégie permet d'implanter simplement différents ”comportements” fonctionnellement identiques. Par exemple, plusieurs algorithmes de tri peuvent être utilisés en fonction d'un contexte. Le motif stratégie est souvent représenté par un diagramme de classe figure C.1.

Dans ce diagramme de classe, la classe Context est associée à l'interface doAction(). Ceci permettra de modifier facilement (à la conception ou à l'exécution) la stratégie. C'est cette capacité évolutive que nous allons essayer de traduire en composants. Nous considerons le cas d'une évolution dynamique car elle nous semble plus proche de la réelle problèmatique à laquelle répond le patron stratégie.

Pour que le changement de stratégie soit transparent au contexte, il suffit de modifier son interface cliente de l'ancienne stratégie à la nouvelle stratégie. Il y a donc nécessairement réorganisation dynamique de l'architecture des composants. C'est précisément cette réorganisation dynamique, et les complications qu'elle peut entrainer que nous allons détailler à présent. Une première implantation naïve de ce patron est illustrée figure C.2.

Dans Fractal, pour modifier les dépendances (liens) entre les composants il faut utiliser les interfaces de contrôle du super-composite (le composite qui contient les composants concernés).

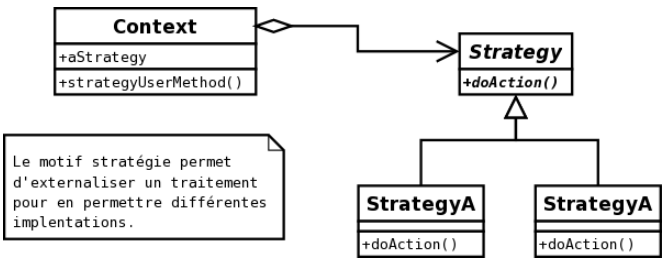

Fig. C.1 – Le motif stratégie - diagramme de classe

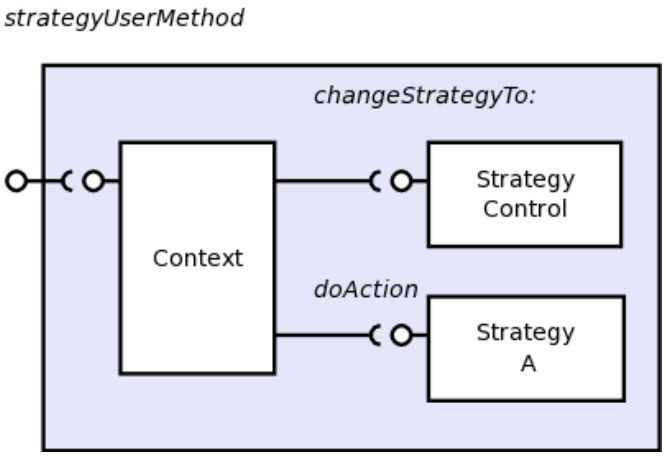

Fig. C.2 – Le motif stratégie : première description naïve en terme de composants

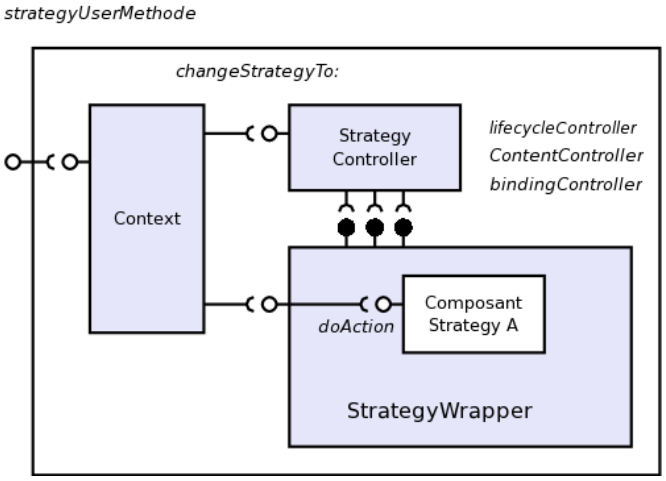

Fig. C.3 – Le motif stratégie : Première proposition

Afin de respecter cette règle, il nous faut “enfermer” la stratégie dans un composant composite (StrategyWrapper). Ce composant va encapsuler le composant Strategy. Cette encapsulation est nécessaire afin de respecter les contraites de Fractal. Une implantation possible est fournie figure C.3.

Le composent StrategyWrapper possède 3 états : *démarré*, *arrété* et *adaptable*.

C.2 Deuxième exemple : La création dynamique de composants

Voici l’exemple d’un fournisseur d’un service. Ils possède plusieurs clients. Chaque client doit s’authentifier pour accéder au service. Le mot de passe et la facture de chaque client sont enre-

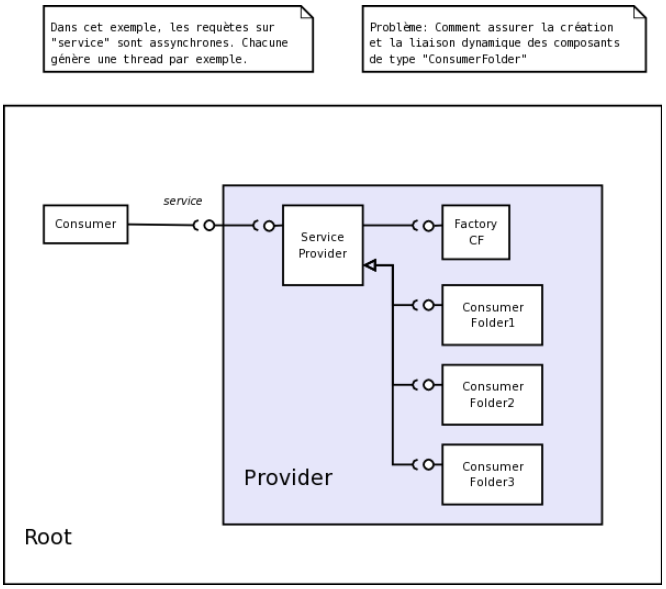

Fig. C.4 – Description du problème

gistrés dans un dossier.

Voici les interfaces :

- Provider>>createAccount:password:creditCard:
- Provider>>accessService:password:
- Provider>>accessFolder:password:
- ConsumerFolder>>account
- ConsumerFolder>>checkPassword:
- ConsumerFolder>>charge:
- ConsumerFolder>>transactions

En fait il s'agit simplement de gérer une interface de cardinalité multiple. L'architecture à composants que l'on souhaite obtenir est représentée figure C.4. C'est une vue idéale et irréaliste selon Fractal. Elle nécessite d'être modifiée pour satisfaire à Fractal. Une première tentative de modélisation est représentée figure C.5. Elle a l'inconvénient d'imposer au composant Root la création et le binding du composant Factory alors même que celui-ci ne devrait pas se soucier du fonctionnement du composant Provider.

Afin de satisfaire au pricipe de "boîte noire" prêt à l'emploi, on va *cacher* le fonctionnement du composant Provider en créeant un composite Pool qui va enfermer les actions de dynamicité. Ainsi la dynamicité nous à contraint à introduire un nouveau composant composite. Il est donc nécessaire de prendre en compte la dynamicité d'un architecture lors de sa conception.

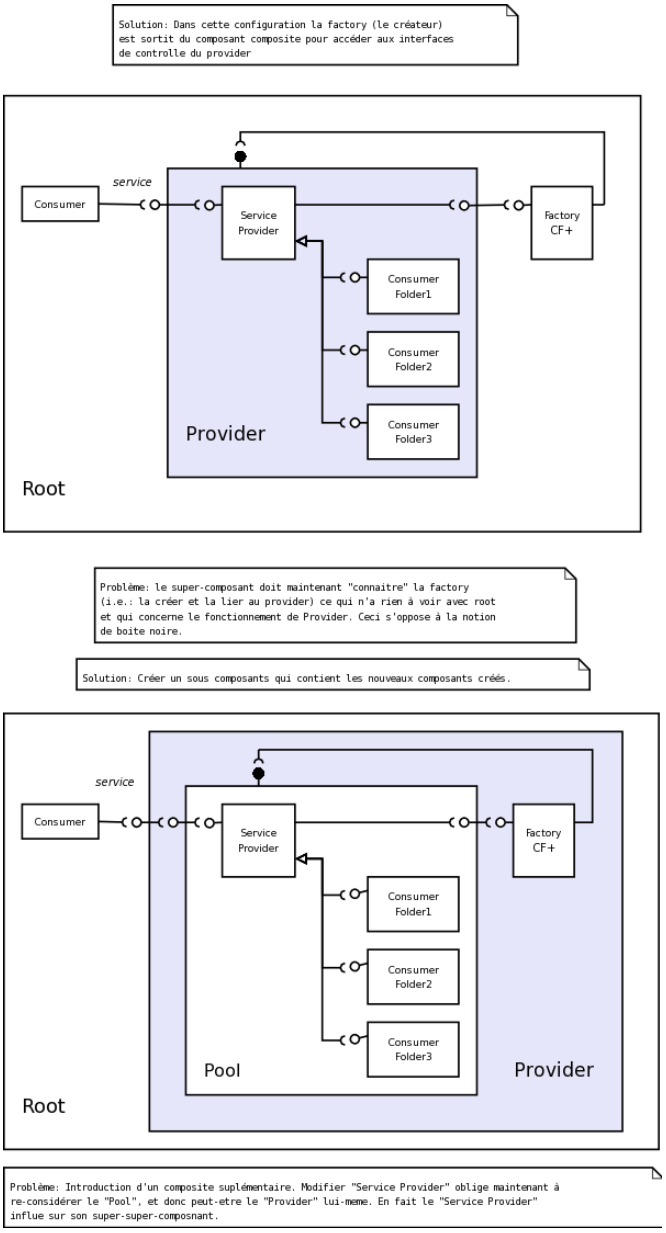

Fig. C.5 – Première solution

Annexe D

Le scénario des agendas

D.1 Introduction

Afin de mettre en œuvre et vérifier nos hypothèses nous avons imaginé un scénario avec plusieurs agendas. Chaque agenda dépend d'un agent qui réside sur une machine particulière. Considérons les trois machines suivantes : une station de travail liée à un réseau fixe, un PDA connecté en wi-fi et un téléphone portable sur un réseau GSM. Imaginons maintenant que l'utilisateur du PDA souhaite prendre rendez-vous avec les utilisateurs des deux autres équipements. Dans cette situation il est évident que les trois machines ne partagent pas les mêmes caractéristiques, aussi il leur sera associé des agents différents. Ce sont ces trois agents que nous allons maintenant décrire.

D.2 Les agents

Agent réactif Dans la mesure où les ressources d'un téléphone mobiles sont limitées, l'agent qu'il devra héberger doit être le plus léger et le moins bavard possible. En effet on peut imaginer que les connections au réseau GSM soient facturées en fonction du trafic. De même la mémoire et l'usage du CPU sont des ressources limitées. Pour répondre à de telles contraintes nous utilisons un agent réactif dont voici les caractéristiques : Un tel agent n'offre pas d'interface de négociation, il permet cependant de consulter l'agenda, et d'y ajouter ou supprimer des rendez-vous. Il n'est pas non plus capable d'exprimer des souhaits, ni de mesurer l'utilité quelconque d'un rendez-vous.

Agent délibératif Les caractéristiques techniques des PDA limitent de moins en moins l'utilisation de tels équipements. Cependant les problèmes de persistance des communications (liés à la mobilité de l'équipement) et son autonomie énergétique en font un appareil non fiable d'un point de vue organisationnel. Aussi l'agent déployé sur un tel appareil devra intégrer ces problématiques. Voici ses caractéristiques : Un tel agent peut participer à des négociations, il ne peut cependant pas les diriger. Il est capable de consulter les autres agents et de prendre des décisions.

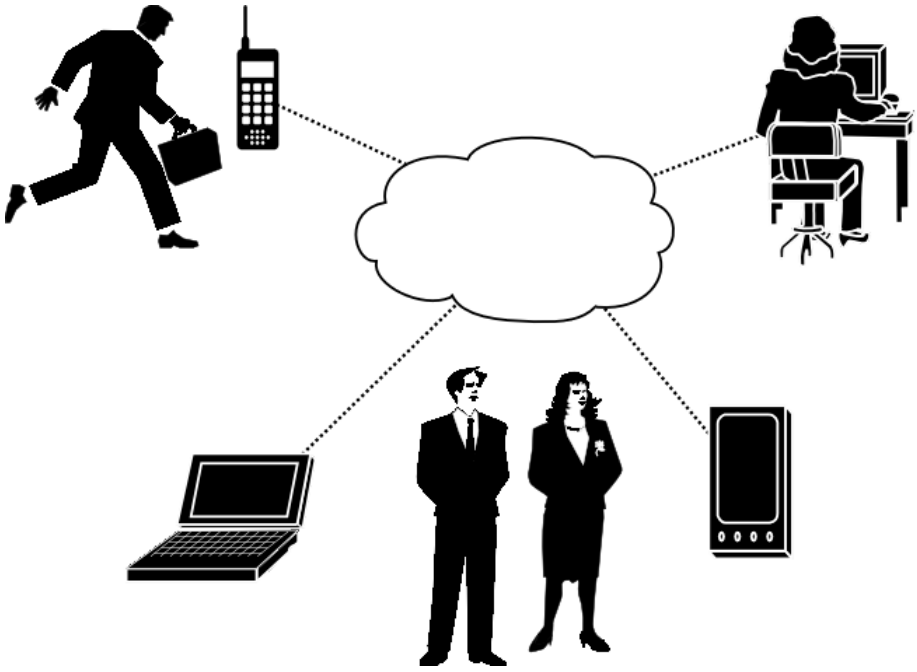

Fig. D.1 – Prise de rendez-vous collective

Agent collaboratif Les ressources disponibles sur les stations de travail actuelles sont compatibles avec l'utilisation d'un agent complexe qui sait réagir, délibérer et collaborer. Ce type d'agent complet peut alors coordonner les interactions entre les équipements moins "riches" de ressources. C'est ce rôle de manager que doit remplir l'agent en plus de son rôle d'agenda bavard. Voici ces caractéristiques : Un tel agent peut pleinement participer à des négociations. Il peut aussi les diriger et consulter les agendas des autres agents. Cet agent est capable de se substituer à un autre agent auquel les ressources ne permettraient pas d'intervenir. Il est évidement capable de prendre des décisions.

D.3 Mise en situation

Afin de continuer l'étude, nous allons décrire rapidement une mise en situation, ce qui devrait permettre de mieux cerner les problématiques. Imaginons alors la situation suivante :

- Pablo souhaite prendre rendez-vous avec Valérie et Nils. Il demande à son PDA de s'arranger avec eux, en spécifiant qu'il souhaiterait les rencontrer en début de semaine prochaine. L'agent associé à Pablo sera appelé agentPDA.
- Valérie est en vacances, mais son ordinateur est resté allumé pour pouvoir être utilisé à distance (comme serveur de fichier par exemple). L'agent associé à Valérie sera appelé agentPC.
- Nils se ballade sur le port de Cherbourg, son téléphone dans la poche. L'agent associé à Nils sera appelé agentGSM.

D.3.1 Détail du scénario

Comment vont s'organiser les agents pour répondre au problème ? En fonction des caractéristiques que nous avons décrit plus haut, voici un scénario possible :

- Tout d'abord l'agentPDA va tenter de résoudre le problème tout seul en consultant les agendas des agents agentPC et agentGSM.
- Ensuite il va s'apercevoir qu'il n'est pas possible de prendre rendez-vous sans déplacer au moins un rendez-vous chez quelqu'un. Il va essayer d'entamer une négociation avec les autres agents.
- À l'appel à la négociation l'agentPC répond "full" et l'agentGSM répond "passive". Ceci signifie que l'agentGSM ne pourra lui-même négocier, mais il accepte la collaboration. Le sens de la réponse "full" de l'agentPC est le suivant : il est d'accord et capable de négocier. Ces ressources peuvent être mises à disposition de la négociation : ainsi il est capable de prendre en charge d'autres agents.
- L'agentPC va prendre la place de l'agentGSM dans la négociation : pour cela il demande à l'agentGSM de lui envoyer son agenda et de l'avertir si celui-ci est modifié le temps de la négociation (C'est l'attribution des rôles dans la négociation).
- L'agentPC est élu "chef" de la négociation : il est maintenant responsable de son bon déroulement. Il récupère les informations communes (les agendas) et les plans-joints disponibles. À partir de ces données, il calcule l'espace de négociation, puis l'envoie aux agents qui participent ; c'est-à-dire agentPDA.
- Chaque agent doit mesurer le coût des plans-joints proposés, sachant que certains plans-joints proposent des déplacements de rendez-vous. Cette mesure de coût s'effectue différemment selon les agents. L'agentPC ne voudra pas prendre la responsabilité de déplacer des rendez-vous de l'agentGSM, les coût seront alors maximum. Mais pour ces propres rendez-vous, il va calculer le coût de chaque déplacement à l'aide d'une "sous négociation" d'autres agents. L'agentPDA quant à lui effectuera seulement des calculs probabiliste sur les chances de déplacement à l'aide des agendas des autres. Ceci lui fournira une fonction de coût.
- La négociation commence et les plans sont tour à tour examinés par le calcul d'un coût total.
- Une fois le plan-joint choisit, instancié et distribué, chaque agent applique le plan local auquel il est associé. Dans cet exemple chacun va enregistrer le rendez-vous finalement négocié. L'agentPC va informer l'agentGSM du rendez-vous, puis déplacer l'autre rendez-vous en conflit.

Bibliographie

[Bou01] Noury Bouraqadi & Thomas Ledoux, Le point sur la programation par aspects *Technique et science informatiques – Synthèse. Volume 20.*

[Bou01] Nicolas Pessemier, Lionel Seinturier & Laurence Duchien, *Une extention de fractal pour l'AOP* Journal francophone de la programmation par aspect – 2004.

[Bro86] Brooks, R. A *A Robust Layered Control System for a Mobile Robot.* IEEE Journal of Robotics and Automation – 1986

[Dal03] Andreas Dalgalarrondo, *À propos de l'autonomie des robots* – Rapport intermédiaire. Centre technique d'Arceuil

[Jun99] Christoph G.Jung, *Therory and Practice of Hybrids Agents.* PhD thesis,. 1999.

[Ml94] Jörg P.Mller, *The Design of Intelligent Agents.* Lecture Notes in Artificial Intelligence 1177.

[Ode04] James Odell, Mariane Nodine & Renato Levy, *A Meta Model for Agent Roles and Groups.* Agent-Oriented Software Engineering (AOSE) V – 2004.

[Pes04] Nicolas Pessemier, Lionel Seinturier, Laurence Duchien & Olivier Barais*Une extention de Fractal pour l'AOP.* JournÃl'e Francophone sur le DÃl'veloppement de Logiciels Par Aspects – 2004

[Rao95] Anand S. Rao & Michael P. Geogeff,*BDI Agents : From Theory to Practive.* Technical notes, 1995.

[Ros96] Michael Rosinus, AlADIn – A language for designing InteRRaP agent.

[Rose95] J. Kenneth Rosenblatt, C. E. T. *Combining multiple goals in a behavior-based architecture* Conf. on Intelligent Robots and Systems (IROS), Pittsburgh, PA, August 1995 (1995).

[Rus91] J.S.Russell & E.Wafald, *Do the rigth thing.* MIT Press, Cambridge Mass, 1991.